Searches and Queries

- In the previous lectures, we have discussed how the Hibernate Session is used to interact with the database.
- Some of the session's methods take query strings in their parameter lists or return Query objects.
- These methods are used to request arbitrary information from the database.
- In order to fully show how they're used, we introduce the Hibernate Query Language (HQL), used to phrase these requests.
- As well as extracting information (with SELECT), HQL can be used to alter the information in the database (with INSERT, UPDATE, and DELETE).

We Cover the Basic Functionality Of HQL in this Lecture

- HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- It is a superset of the JPQL, the Java Persistence Query Language; a JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries.
- HQL is a language with its own syntax and grammar.
- Ultimately, your HQL queries are translated by Hibernate into conventional SQL queries; Hibernate also provides an API that allows you to directly issue SQL queries.
- Note: Hibernate's query facilities do not allow you to alter the database structure.

Hibernate Query Language (HQL)

- While most ORM tools and object databases offer an object query language, Hibernate's HQL stands out as complete and easy to use.

- Although you can use SQL statements directly with Hibernate, we recommend that you use HQL (or criteria) whenever possible to avoid database portability hassles, as well as to take advantage of Hibernate's SQL-generation and caching strategies.

- In addition to its technical advantages over traditional SQL, HQL is a more compact query language than SQL because it can make use of the relationship information defined in the Hibernate mappings.

Not every developer trusts Hibernate's generated SQL to be perfectly optimized.

- If you do encounter a performance bottleneck in your queries, we recommend that you use SQL tracing on your database during performance testing of your critical components.
- If you see an area that needs optimization, first try to optimize using HQL, and only later drop into native SQL.
- Hibernate provides statistics information through a JMX MBean, which you can use for analyzing Hibernate's performance.
- Hibernate's statistics also give you insight into how caching is performing.

Hibernate Console Plug-in

- If you would like to execute HQL statements through a GUI-based tool, the Hibernate team provides a Hibernate console for Eclipse in the Hibernate Tools subproject.

- This console is a plug-in for recent versions of Eclipse.

Syntax Basics

- HQL was inspired by SQL and is the inspiration for the Java Persistence Query Language (JPQL).
- The JPQL specification is included in the standard for JPA available from the Java Community Process web site (www.jcp.org/en/jsr/detail?id=220).

- HQL's syntax is defined as an ANTLR grammar; the grammar files are included in the grammar directory of the Hibernate core download. (ANTLR is a tool for building language parsers.)

- As the ANTLR grammar files are somewhat cryptic, and as not every statement that is permissible according to the ANTLR grammar's rules can be used in Hibernate, we outline the syntax for the four fundamental HQL operations.
- Note that the following descriptions of syntax are not comprehensive; there are some deprecated or more obscure usages (particularly for SELECT statements) that are not covered here.

UPDATE

UPDATE alters the details of existing objects in the database. In-memory entities, managed or not, will not be updated to reflect changes resulting from issuing UPDATE statements. Here's the syntax of the UPDATE statement:

```
UPDATE [VERSIONED]
   [FROM] path [[AS] alias] [, ...]
   SET property = value [, ...]
   [WHERE logicalExpression]
```

The fully qualified name of the entity or entities is path. The alias names may be used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.

VERSIONED means that the update will update time stamps, if any, that are part of the entity being updated.

The property names are the names of properties of entities listed in the FROM path.

The syntax of logical expressions is discussed later, in the "Using Restrictions with HQL" section.

An example of the update in action might look like this:

```
Query query=session.createQuery("update Person set creditscore=:creditscore where name=:name");
query.setInteger("creditscore", 612);
query.setString("name", "John Q. Public");
int modifications=query.executeUpdate();
```

DELETE

DELETE removes the details of existing objects from the database. In-memory entities will not be updated to reflect changes resulting from DELETE statements. This also means that Hibernate's cascade rules will not be followed for deletions carried out using HQL. However, if you have specified cascading deletes at the database level (either directly or through Hibernate, using the @OnDelete annotation), the database will still remove the child rows. This approach to deletion is commonly referred to as "bulk deletion," since it is the most efficient way to remove large numbers of entities from the database. Here's the syntax of the DELETE statement:

```
DELETE
   [FROM] path [[AS] alias]
   [WHERE logicalExpression]
```

The fully qualified name of an entity is path. The alias names may be used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.

```
Query query=session.createQuery("delete from Person where accountstatus=:status");
query.setString("status", "purged");
int rowsDeleted=query.executeUpdate();
```

INSERT

An HQL INSERT cannot be used to directly insert arbitrary entities—it can only be used to insert entities constructed from information obtained from SELECT queries (unlike ordinary SQL, in which an INSERT command can be used to insert arbitrary data into a table, as well as insert values selected from other tables). Here's the syntax of the INSERT statement:

```
INSERT
   INTO path ( property [, ...])
   select
```

The name of an entity is path. The property names are the names of properties of entities listed in the FROM path of the incorporated SELECT query.

The select query is an HQL SELECT query (as described in the next section).

As this HQL statement can only use data provided by an HQL select, its application can be limited. An example of copying users to a purged table before actually purging them might look like this:

```
Query query=session.createQuery("insert into purged_users(id, name, status) "+
    "select id, name, status from users where status=:status");
query.setString("status", "purged");
int rowsCopied=query.executeUpdate();
```

SELECT

An HQL SELECT is used to query the database for classes and their properties. As noted previously, this is very much a summary of the full expressive power of HQL SELECT queries; however, for more complex joins and the like, you may find that using the Criteria API described in the next chapter is more appropriate. Here's the syntax of the SELECT statement:

```
[SELECT [DISTINCT] property [, ...]]
   FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]
   WHERE logicalExpression
   GROUP BY property [, ...]
   HAVING logicalExpression
   ORDER BY property [ASC | DESC] [, ...]
```

The fully qualified name of an entity is `path`. The `alias` names may be used to abbreviate references to specific entities or their properties, and must be used when property names used in the query would otherwise be ambiguous.

The `property` names are the names of properties of entities listed in the `FROM` path.

If `FETCH ALL PROPERTIES` is used, then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).

When the properties listed consist only of the names of aliases in the `FROM` clause, the `SELECT` clause can be omitted in HQL. If you are using the JPA with JPQL, one of the differences between HQL and JPQL is that the `SELECT` clause is required in JPQL.

Named Queries

- Hibernate (and JPA) provide named queries.
- Named queries are created via class-level annotations on entities; normally, the queries apply to the entity in whose source file they occur, but there's no absolute requirement for this to be true.
- Named queries are created with the @NamedQueries annotation, which contains an array of @NamedQuery sets; each has a query and a name.

Let us create an object model we can use as an example

- Our object model will contain products and suppliers; it will also contain a specialized product (“Software”) that adds an attribute to Product.
- We will to eliminate some of the boilerplate from the source code—namely, constructors, mutators, accessors, equals(), hashCode(), and toString().
- This object model structure is shown in the following slides.

The Supplier Entity, with Boilerplate Removed


```
@Entity
public class Supplier implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    Integer id;
    @Column(unique = true)
    @NotNull
    String name;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true,
            mappedBy = "supplier", targetEntity = Product.class)
    List<Product> products = new ArrayList<>();
}
```

The Basic Product Entity, with Boilerplate Removed


```
@Entity
public class Product implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    Integer id;
    @ManyToOne(optional = false, fetch = FetchType.LAZY)
    Supplier supplier;
    @Column
    @NotNull
    String name;
    @Column
    @NotNull
    String description;
    @Column
    @NotNull
    Double price;
}
```

The Software Entity, with Boilerplate Removed


```
@Entity
public class Software extends Product implements Serializable {
    @Column
    @NotNull
    String version;
}
```

Adding a named query is as simple as adding an annotation to one of the entities. For example, if we wanted to add a named query to retrieve all `Supplier` entities, we could do so by adding a `@NamedQuery` annotation to *any* of the entities, although it makes the most sense to put the query in `Supplier`'s source code.

```
@NamedQuery(name = "supplier.findAll", query = "from Supplier s")
```

You can group queries together by adding a `@NamedQueries` annotation, and then embedding the named queries in an array as part of that annotation. This would look like this:

```
@NamedQueries({
        @NamedQuery(name = "supplier.findAll", query = "from Supplier s"),
        @NamedQuery(name = "supplier.findByName",
                query = "from Supplier s where s.name=:name"),
})
```

Getting Named Query


```
Query query = session.getNamedQuery("supplier.findAll");
List<Supplier> suppliers = query.list();
```

- Of course, we could create queries on the fly, as we've done in the lab sessions.

- another example (for the sake of completeness) for querying all Products:

```
Query query = session.createQuery("from Product");
query.setComment("This is only a query for product");
List<Product> products = query.list();
```

- *You can also add a comment to the SQL that Hibernate creates, which is useful for tracing which HQL statements correspond to which SQL statements.*

More on HQL

- Like all SQL syntax, you can write from in lowercase or uppercase (or mixed case).
- However, any Java classes or properties that you reference in an HQL query have to be specified in the proper case.
- For example, when you query for instances of a Java class named Product, the HQL query `from Product` is the equivalent of `FROM Product`.
- However, the HQL query `from product` is not the same as the HQL query `from Product`.
- Because Java class names are case-sensitive, Hibernate is case-sensitive about class names as well.

Creating the Query Object

- The createQuery() method takes a valid HQL statement and returns an org.hibernate.Query object.
- The Query class provides methods for returning the query results as a Java List, as an Iterator, or as a unique result.
- Other functionality includes named parameters, results scrolling, JDBC fetch sizes, and JDBC timeouts.
- Official Hibernate JavaDoc for Query:
http://docs.jboss.org/hibernate/orm/5.1/javadocs/org/hibernate/Query.html

Interface Query

All Superinterfaces:

BasicQueryContract

All Known Subinterfaces:

SQLQuery

All Known Implementing Classes:

AbstractQueryImpl, CollectionFilterImpl, QueryImpl, SQLQueryImpl

---

```
public interface Query
extends BasicQueryContract
```

An object-oriented representation of a Hibernate query. A `Query` instance is obtained by calling `Session.createQuery()`. This interface exposes some extra functionality beyond that provided by `Session.iterate()` and `Session.find()`:

- a particular page of the result set may be selected by calling `setMaxResults()`, `setFirstResult()`
- named query parameters may be used
- the results may be returned as an instance of `ScrollableResults`

Named query parameters are tokens of the form `:name` in the query string. A value is bound to the `integer` parameter `:foo` by calling

```
setParameter("foo", foo, Hibernate.INTEGER);
```

for example. A name may appear multiple times in the query string.

JDBC-style `?` parameters are also supported. To bind a value to a JDBC-style parameter use a set method that accepts an `int` positional argument (numbered from zero, contrary to JDBC).

You may not mix and match JDBC-style parameters and named parameters in the same query.

Queries are executed by calling `list()`, `scroll()` or `iterate()`. A query may be re-executed by subsequent invocations. Its lifespan is, however, bounded by the lifespan of the `Session` that created it.

Logging and Commenting the Underlying SQL

- Hibernate can output the underlying SQL behind your HQL queries into your application's log file.
- This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted.
- You can run the SQL that Hibernate generates directly against your database in the database's query analyzer at a later date to determine the causes of the problem.
- This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

Logging the SQL

The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the `show_sql` property. Set this property to `true` in your `hibernate.cfg.xml` configuration file, and Hibernate will output the SQL into the logs. You do not need to enable any other logging settings, although setting logging for Hibernate to debug also outputs the generated SQL statements, along with a lot of other verbiage.

After enabling SQL output in Hibernate, you should rerun the previous example. Here is the generated SQL statement for the HQL statement `from Product`:

```
Hibernate: /* This is only a query for product */ select product0_.id as id1_0_, product0_.
description as descript2_0_, product0_.name as name3_0_, product0_.price as price4_0_, product0_.
supplier_id as supplier5_0_, product0_1_.version as version1_1_, case when product0_1_.id is not
null then 1 when product0_.id is not null then 0 end as clazz_ from Product product0_ left outer
join Software product0_1_ on product0_.id=product0_1_.id
```

As an aside, remember that the `Software` class inherits from `Product`, which complicates Hibernate's generated SQL for this simple query. When we select all objects from our simple `Supplier` class, the generated SQL for the HQL query `from Supplier` is much simpler:

```
select supplier0_.id as id1_2_, supplier0_.name as name2_2_ from Supplier supplier0_
```

When you look in your application's output for the Hibernate SQL statements, they will be prefixed with "`Hibernate:`". The previous SQL statement would look like this:

```
Hibernate: /* named HQL query supplier.findAll */ select supplier0_.id as id1_2_, supplier0_.name as
name2_2_ from Supplier supplier0_
```

Commenting the Generated SQL

Tracing your HQL statements through to the generated SQL can be difficult, so Hibernate provides a commenting facility on the `Query` object that lets you apply a comment to a specific query. The `Query` interface has a `setComment()` method that takes a `String` object as an argument, as follows:

```
public Query setComment(String comment)
```

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the `setComment()` method. You will also need to set a Hibernate property, `hibernate.use_sql_comments`, to `true` in your Hibernate configuration.[4] If you set this property but do not set a comment on the query programmatically, Hibernate will include the HQL used to generate the SQL call in the comment. We find this to be very useful for debugging HQL.

Use commenting to identify the SQL output in your application's logs if SQL logging is enabled. For instance, if we add a comment to this example, the Java code would look like this:

```
String hql = "from Supplier";
Query query = session.createQuery(hql);
query.setComment("My HQL: " + hql);
List results = query.list();
```

The output in your application's log will have the comment in a Java-style comment before the SQL:

```
Hibernate: /*My HQL: from Supplier*/ select supplier0_.id as id, supplier0_.name ➥
as name2_ from Supplier supplier0_
```

This can be useful for identifying SQL in your logs, especially because the generated SQL is a little difficult to follow when you are scanning large quantities of it. (Running the example code from this test class serves as a great example; it's hundreds of lines' worth of output.)

The from Clause and Aliases

- We have already discussed the basics of the from clause in HQL.
- The most important feature to note is the *alias*.
- Hibernate allows to assign aliases to the classes in your query with the as clause.
- Use the aliases to refer back to the class inside the query.
- For instance, our previous simple example would be the following:

```
from Product as p
```

- or the following:

```
from Product as product
```

- You'll see either alias-naming convention in applications.
- The as keyword is optional—you can also specify the alias directly after the class name, as follows:

```
from Product product
```

Using Package Names

- If you need to fully qualify a class name in HQL, just specify the package and class name.
- Hibernate will take care of most of this behind the scenes, so you really need this only if you have classes with duplicate names in your application.
- If you have to do this in Hibernate, use syntax such as the following:

```
from myPackage.model.Product
```

SELECT Clause

- The from clause is very basic and useful for working directly with objects.
- However, if you want to work with the object's properties without loading the full objects into memory, you must use the select clause.

The select Clause and Projection

The `select` clause provides more control over the result set than the `from` clause. If you want to obtain the properties of objects in the result set, use the `select` clause. For instance, we could run a projection query on the products in the database that only returned the names, instead of loading the full object into memory, as follows:

```
select product.name from Product product
```

The result set for this query will contain a `List` of Java `String` objects. Additionally, we can retrieve the prices and the names for each product in the database, like so:

```
select product.name, product.price from Product product
```

This result set contains a `List` of `Object` arrays—each array represents one set of properties (in this case, a name and price pair).

If you're only interested in a few properties, this approach can allow you to reduce network traffic to the database server and save memory on the application's machine.

Using Restrictions with HQL

As with SQL, you use the `where` clause to select results that match your query's expressions. HQL provides many different expressions that you can use to construct a query. In the HQL language grammar, there are *many* possible expressions, including:

- *Logic operators*: `OR`, `AND`, `NOT`
- *Equality operators*: `=`, `<>`, `!=`, `^=`
- *Comparison operators*: `<`, `>`, `<=`, `>=`, `like`, `not like`, `between`, `not between`
- *Math operators*: `+`, `-`, `*`, `/`
- *Concatenation operator*: `||`
- *Cases*: Case when `<logical expression>` then `<unary expression>` else _`<unary expression>` end
- *Collection expressions*: `some`, `exists`, `all`, `any`

In addition, you may also use the following expressions in the `where` clause:

- *HQL named parameters*:, such as `:date`, `:quantity`
- *JDBC query parameter*: ?
- *Date and time SQL-92 functional operators*: `current_time()`, `current_date()`, `current_timestamp()`
- *SQL functions (supported by the database)*: `length()`, `upper()`, `lower()`, `ltrim()`, `rtrim()`, etc.

Using Named Parameters

- Hibernate supports named parameters in its HQL queries.
- This makes writing queries that accept input from the user easy—and you do not have to defend against SQL injection attacks.
- You could escape the user's input yourself for every query, but it is much less of a security risk if you let Hibernate manage all of your input with named parameters.

Note SQL injection is an attack against applications that create SQL directly from user input with string concatenation. For instance, if we accept a name from the user through a web application form, then it would be very bad form to construct an SQL (or HQL) query like this:

```
String sql = "select p from products where name = '" + name + "'";
```

A malicious user could pass a name to the application that contained a terminating quote and semicolon, followed by another SQL command (such as `delete from products`) that would let the user do whatever he wanted. He would just need to end with another command that matched the SQL statement's ending quote. This is a very common attack, especially if the malicious user can guess details of your database structure.[6]

Named Parameters

- Hibernate's named parameters are similar to the JDBC query parameters (?) you are already be familiar with, but Hibernate's parameters are less confusing.
- It is also more straightforward to use Hibernate's named parameters if you have a query that uses the same parameter in multiple places.
- When using JDBC query parameters, any time you add, change, or delete parts of the SQL statement, you need to update your Java code that sets its parameters, because the parameters are indexed based on the order in which they appear in the statement.
- Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

Named Parameter Example

- A simple example of named parameters:

```
String hql = "from Product where price > :price";
Query query = session.createQuery(hql);
query.setDouble("price",25.0);
List results = query.list();
```

- When the value to be provided will be known only at run time, you can use some of HQL's object-oriented features to provide objects as values for named parameters.

Paging Through the Result Set

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the `Query` interface for paging: `setFirstResult()` and `setMaxResults()`, just as with the `Criteria` interface. The `setFirstResult()` method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the `setMaxResults()` method. Your HQL is unchanged—you need only to modify the Java code that executes the query. Excuse our tiny dataset for this trivial example of pagination:

```
Query query = session.createQuery("from Product");
query.setFirstResult(1);
query.setMaxResults(2);
List results = query.list();
displayProductsList(results);
```

You can change the numbers around and play with the pagination. If you turn on SQL logging, you can see which SQL commands Hibernate uses for pagination. For the open-source HSQLDB database, Hibernate uses `top` and `limit`. For other databases, Hibernate uses the appropriate commands for pagination. For instance, Microsoft SQL Server does not support the `limit` command, so Hibernate uses only the `top` command. If your application is having performance problems with pagination, this can be very helpful for debugging.

If you only have one result in your HQL result set, Hibernate has a shortcut method for obtaining just that object.

Obtaining a Unique Result from an HQL Query

- HQL's Query interface provides a uniqueResult() method for obtaining just one object from an HQL query.
- Although your query may yield only one object, you may also use the uniqueResult() method with other result sets if you limit the results to just the first result.
- The uniqueResult() method on the Query object returns a single object, or null if there are zero results.
- If there is more than one result, then the uniqueResult() method throws a NonUniqueResultException.

Example

- The following example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

```
String hql = "from Product where price>25.0";
Query query = session.createQuery(hql);
query.setMaxResults(1);
Product product = (Product) query.uniqueResult();
//test for null here if needed
```

- Unless your query returns one or zero results, the uniqueResult() method will throw a NonUniqueResultException exception.

- Do not expect Hibernate just to pick off the first result and return it—either set the maximum results of the HQL query to 1, or obtain the first object from the result list.

Sorting Results with the order by Clause

- To sort your HQL query's results, you will need to use the order by clause.
- You can order the results by any property on the objects in the result set: either ascending (asc) or descending (desc).
- You can use ordering on more than one property in the query, if you need to
- A typical HQL query for sorting results looks like this:

```
from Product p where p.price > 25.0 order by p.price desc
```

Sorting by More Than One Property

- If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas.
- For example, you could sort by product price and the supplier's name:

```
from Product p order by p.supplier.name asc, p.price asc
```

- HQL is more straightforward for ordering than the equivalent approach using the Criteria Query API.

Associations

- Associations allow you to use more than one class in an HQL query, just as SQL allows you to use joins between tables in a relational database.
- You add an association to an HQL query with the join clause.
- Hibernate supports five different types of joins:
  - inner join
  - cross join
  - left outer join
  - right outer join
  - full outer join.

Example

- If you use cross join, just specify both classes in the from clause

  from Product p, Supplier s

- For the other joins, use a join clause after the from clause. Specify the type of join, the object property to join on, and an alias for the other class

- You can use inner join to obtain the supplier for each product, and then retrieve the supplier name, product name, and product price, as so:

  select s.name, p.name, p.price from Product p inner join p.supplier as s

- You can retrieve the objects using similar syntax:

  from Product p inner join p.supplier as s

We used aliases in these HQL statements to refer to the entities in our query expressions. These are particularly important in queries with associations that refer to two different entities with the same class—for instance, if we are doing a join from a table back to itself. Commonly, these types of joins are used to organize tree data structures.

Notice that Hibernate does not return `Object` objects in the result set; instead, Hibernate returns `Object` arrays in the results.[7] You will have to access the contents of the `Object` arrays to get the `Supplier` and the `Product` objects.

If you would like to start optimizing performance, you can use one query to ask Hibernate to fetch the associated objects and collections for an object. If you were using lazy loading with Hibernate, the objects in the collection would not be initialized until you accessed them. If you use `fetch` on a join in your query, you can ask Hibernate to retrieve the objects in the collection at the time the query executes. Add the `fetch` keyword after the `join` in the query, like so:

```
from Supplier s inner join fetch s.products as p
```

When you use `fetch` for a query like this, Hibernate will return only the `Supplier` objects, not the `Product` objects. This is because you are specifying the join, so Hibernate knows which objects to fetch (instead of using lazy loading). If you need to get the `Product` objects, you can access them through the associated `Supplier` object. You cannot use the properties of the `Product` objects in expressions contained in the `where` clause. Use of the `fetch` keyword overrides any settings you have in the mapping file for object initialization.

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. The `count(...)` method returns the number of times the given column name appears in the result set. You may use the `count(*)` syntax to count all the objects in the result set, or `count(product.name)` to count the number of objects in the result set with a `name` property. Here is an example using the `count(*)` method to count all products:

```
select count(*) from Product product
```

The `distinct` keyword only counts the unique values in the row set—for instance, if there are 100 products, but 10 have the same price as another product in the results, then a `select count(distinct product.price) from Product product` query would return 90. In our database, the following query will return 2, one for each supplier:

```
select count(distinct product.supplier.name) from Product product
```

If we removed the `distinct` keyword, it would return 5, one for each product.

All of these queries return a `Long` object in the list. You could use the `uniqueResult()` method here to obtain the result.

The aggregate functions available through HQL include the following:

- `avg(`*property name*`)`: The average of a property's value
- `count(`*property name* or `*)`: The number of times a property occurs in the results
- `max(`*property name*`)`: The maximum value of the property values
- `min(`*property name*`)`: The minimum value of the property values
- `sum(`*property name*`)`: The sum total of the property values

Bulk Updates and Deletes with HQL

The Query interface contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.[8] The executeUpdate() method returns an int that contains the number of rows affected by the update or delete, as follows:

```
public int executeUpdate() throws HibernateException
```

HQL updates look as you would expect them to, being based on SQL UPDATE statements. Do not include an alias with the update; instead, put the set keyword right after the class name, as follows:

```
String hql = "update Supplier set name = :newName &#x00E5;
where name = :name";

Query query = session.createQuery(hql);
query.setString("name","SuperCorp");
query.setString("newName","MegaCorp");
int rowCount = query.executeUpdate();
System.out.println("Rows affected: " + rowCount);

//See the results of the update
query = session.createQuery("from Supplier");
List results = query.list();
```

After carrying out this query, any supplier previously named SuperCorp will be named MegaCorp. You may use a where clause with updates to control which rows get updated, or you may leave it off to update all rows. Notice that we printed out the number of rows affected by the query. We also used named parameters in our HQL for this bulk update.

Bulk deletes work in a similar way. Use the delete from clause with the class name you would like to delete from. Then use the where clause to narrow down which entries in the table you would like to delete. Use the executeUpdate() method to execute deletes against the database as well.

Be careful when you use bulk delete with objects that are in relationships. Hibernate will not know that you removed the underlying data in the database, and you can get foreign key integrity errors.

Caution

- Using bulk updates and deletes in HQL works almost the same as in SQL, so keep in mind that these are powerful and can erase the data in your tables if you make a mistake with the where clause.

Using Native SQL

Although you should probably use HQL whenever possible, Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application. We discuss stored procedures and other database-specific integration solutions in Appendix A. Rather than just providing an interface to the underlying JDBC connection, like other Java ORM tools, Hibernate provides a way to define the entity (or join) that the query uses. This makes integration with the rest of your ORM-oriented application easy.

You can modify your SQL statements to make them work with Hibernate's ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {`objectname.*`}, or you can specify the aliases directly with {`objectname.property`}. Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so that you select the right properties off the table.

Underlying Hibernate's native SQL support is the `org.hibernate.SQLQuery` interface, which extends the `org.hibernate.Query` interface already discussed. Your application will create a native SQL query from the session with the `createSQLQuery()` method on the `Session` interface.

```
public SQLQuery createSQLQuery(String queryString) throws HibernateException
```

After you pass a string containing the SQL query to the `createSQLQuery()` method, you should associate the SQL result with an existing Hibernate entity, a join, or a scalar result. The `SQLQuery` interface has `addEntity()`, `addJoin()`, and `addScalar()` methods. For the entities and joins, you can specify a lock mode, which we discussed in Chapter 8. The `addEntity()` methods take an alias argument and either a class name or an entity name. The `addJoin()` methods take an alias argument and a path to join.

Using native SQL with scalar results is the simplest way to get started with native SQL. Our Java code looks like this:

```
String sql = "select avg(product.price) as avgPrice from Product product";
SQLQuery query = session.createSQLQuery(sql);
query.addScalar("avgPrice",Hibernate.DOUBLE);
List results = query.list();
```

Because we did not specify any entity aliases, Hibernate executes exactly the same SQL that we passed through:

```
select avg(product.price) as avgPrice from Product product
```

The SQL is regular SQL (we did not have to do any aliasing here). We created an `SQLQuery` object, and then added a scalar mapping with the built-in `double` type (from the `org.hibernate._Hibernate` class). We needed to map the `avgPrice` SQL alias to the object type. The results are a `List` with one object—a `Double`.

A bit more complicated than the previous example is the native SQL that returns a result set of objects. In this case, we will need to map an entity to the SQL query. The entity consists of the alias we used for the object in the SQL query and its class. For this example, we used our `Supplier` class:

```
String sql = "select {supplier.*} from Supplier supplier";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity("supplier", Supplier.class);
List results = query.list();
```

Hibernate modifies the SQL and executes the following command against the database:

```
select Supplier.id as id0_, Supplier.name as name2_0_ from Supplier supplier
```

The special aliases allow Hibernate to map the database columns back to the object properties.